SC

Generated by Doxygen 1.5.7.1

Sun Mar 14 19:46:12 2010

Contents

Chapter 1

File Index

1.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 2

File Documentation

2.1 sc.c File Reference

```
#include "sc.h"
```

Functions

- void selectCid ()

  Computing the id of next thread to be dispatched.

- void vec2names (char *prefix, char *suffix, bitvector ids, const char *names[ ])

  Tracing routines.

- int main ()

  The main program.

2.1.1 Detailed Description

Main file for using SyncChart C macros

See README.txt for general information. See LICENSE.txt for licensing information. For further information, see http://www.informatik.uni-kiel.de/rtsys/sc/ .

Author:

Reinhard v. Hanxleden, rvh@informatik.uni-kiel.de

2.1.2 Function Documentation

2.1.2.1 int main (void)

The main program.

Returns 0 iff outputs generated by program match reference trace

2.1.2.2 void selectCid ()

Computing the id of next thread to be dispatched.

Functions defined in sc.c.

Uses obvious algorithm, run time linear in position of highest bit. Note that there are also alternatives that run logarithmic to bit vector size. See eg http://graphics.stanford.edu/~seander/bithacks.html#IntegerLog . Which is actually faster depends on application.

2.2 sc.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
```

Defines

- #define **MAX**(a, b) a > b ? a : b
- #define **MIN**(a, b) a < b ? a : b
- #define **_SC_SWITCHLOGIC**
- #define _idMax 8∗sizeof(threadvector)

  Number of threads.

- #define **clearPC**(id)
- #define **initPC**(p, label)
- #define **setPC**(id, label)
- #define _declState static labeltype _state;

  Dispatcher.

- #define **_BEGIN_SWITCH**
- #define **_END_SWITCH1** }
- #define **_END_SWITCH2** }
- #define **_case** case
- #define **_break** break
- #define **_goto**(label) do { _state = label; goto _L_SWITCH; } while (0)
- #define **_deref**(label) label
- #define **_ref**(label) label
- #define **_setStateInit** _state = _L_INIT;
- #define **selectCid_**() selectCid()
- #define **dispatch**() selectCid_(); _goto(_deref(_pc[_cid]))
- #define u2b(u) (1 << u)

  Encoding of signal/thread 'u' (some non-negative int) in bitvector.

- #define enable(id)

  Thread enabling/disabling.

- #define **enableInit**(id)
- #define **disable**(id) enabled &= ∼u2b(id)
- #define **disableSet**(idset) enabled &= ∼idset
- #define **isEnabled**(id) (enabled & u2b(id))
- #define **isEnabledNotOnly**(id) (enabled != u2b(id))
- #define **isEnabledNoneOf**(idset) ((enabled & idset) == 0)
- #define **isEnabledAnyOf**(idset) ((enabled & idset) != 0)
- #define activate(id) active |= u2b(id)

  Thread (de-)activation.

- #define **deactivate**(id) active &= ∼u2b(id)
- #define **deactivateSet**(idset) active &= ∼idset
- #define **isActive**(id) (active & u2b(id))

- #define **_TickEnd** 0
- #define RESET()

  Reset automaton. Should be called before first call of tick function.

- #define _SC_ERROR_DETECT_NORESET

  Start a tick (an instant). 'p' denotes the main thread.

- #define **TICKSTART**(p)
- #define TICKEND

  Complete a tick.

- #define dispatch_ goto _L_DISPATCH
- #define **mergedDispatch**
- #define _CONCAT_helper(a, b) a ## b

  Construct a label, of the form "_L'line in source file'".

- #define **_CONCAT**(a, b) _CONCAT_helper(a, b)
- #define **__LABELL__** _CONCAT(_LL, __LINE__)
- #define **__LABEL__** __LINE__
- #define PAUSE

  Pause a thread, resume at subsequent statement.

- #define PAUSEG(label)

  Shorthand for 'PAUSE; GOTO(label)'.

- #define PAUSEG_(label)

  Helper function (if/else-unsafe).

- #define HALT

  Shorthand for 'label: PAUSE; GOTO(label)'.

- #define SUSPEND(cond)

  Suspend current thread if 'cond' is true.

- #define SUSPENDG(label)

  Suspend current thread, goto 'label'.

- #define TRANS(label)

  Transition to 'label', kill descendant threads (implements abortion).

- #define ABORT

  Abort (terminate) descendant threads.

- #define ABORT_

  Helper function (if/else-unsafe).

- #define TERM

  Terminate a thread.

- #define TERM_ goto _L_TERM

Helper function (if/else-unsafe).

- #define TERM_ goto _L_TERM

  Helper function (if/else-unsafe).

- #define FORK(label, p)

  Spawn a thread at 'label', with priority 'p'.

- #define FORK_(label, p)

  Helper function (if/else-unsafe).

- #define FORKE(label)

  Denote parent thread, starting at 'label'.

- #define FORKE_(label)

  Helper function (if/else-unsafe).

- #define JOINELSE(elselabel)

  Join completed child threads.

- #define JOINELSEG(thenlabel, elselabel)

  Shorthand for 'JOINELSE(elselabel); GOTO(thenlabel)'.

- #define JOINELSEG_(thenlabel, elselabel)

  Helper function (if/else-unsafe).

- #define JOIN

  Shorthand for 'elselabel: JOINELSE(elselabel)'. Join completed child threads.

- #define PRIO(p)

  Change priority of a thread, from '_cid' to 'p'.

- #define PRIOG(p, label)

  Shorthand for 'PRIO(p); GOTO(label)'.

- #define _SC_ERROR_DETECT_PRIO(p)

  Check whether PRIO tries to acquire priority that is already in use.

- #define PRIO_(p)

  Helper function (if/else-unsafe) to change priority of a thread, from '_cid' to 'p'.

- #define PRIOG_(p, label)

  Helper function (if/else-unsafe).

- #define PPAUSEG(p, label)

  Efficient shorthand for 'PRIO(p); PAUSE; GOTO(label)'.

- #define PPAUSEG_(p, label)

  Helper function (if/else-unsafe).

- #define PPAUSE(p)

  Efficient shorthand for 'PRIO(p); PAUSE'.

- #define JPPAUSEG(p, thenlabel, elselabel)

  Efficient shorthand for 'JOINELSE(label); GOTO thenlabel; label: PRIO(p); PAUSE; GOTO(elselabel)'.

- #define JPPAUSEG_(p, thenlabel, elselabel)

  Helper function (if/else-unsafe).

- #define JPPAUSE(p, elselabel)

  Shorthand for 'JOINELSE(label); GOTO thenlabel; label: PRIO(p); PAUSE; GOTO(elselabel); thenlabel:'.

- #define SIGNAL(s)

  Initialize a local signal (handles reincarnation).

- #define _checkTickInit _presence_tested = 0;

  Check whether an emitted signal has already been checked for presence.

- #define **_checkPRESENT**(s) _presence_tested |= u2b(s);
- #define **_checkPRESENTc**(s) _presence_tested |= u2b(s),
- #define **_checkEMIT**(s)
- #define EMIT(s)

  Emission of a pure signal 's'.

- #define SUSTAIN(s)

  Sustain a pure signal 's'.

- #define PRESENT(s)

  Test for presence of signal 's' (predicate).

- #define PRESENTELSE(s, label)

  Test for presence of signal 's' (control flow op).

- #define PRESENTEMIT(s, t)

  If signal 's' is present, emit 't'.

- #define AWAITI(s)

  Await (immediately) signal 's'.

- #define AWAIT(s)

  Await (non-immediately) signal 's'.

- #define **_declindex**
- #define **_setValInit**
- #define EMITINT(s, val)

  Emission of a valued signal 's', type integer.

- #define EMITINTMUL(s, val)

  *Emission of a valued signal 's', type integer, combined with * .*

- #define EMITINTADD(s, val)

  Emission of a valued signal 's', type integer, combined with + .

- #define EMITINTMAX(s, val)

  Emission of a valued signal 's', type integer, combined with the maximum function .

- #define EMITINTMIN(s, val)

  Emission of a valued signal 's', type integer, combined with the minimum function .

- #define VAL(s)

  Retrieve value of signal 's'.

- #define VALREG(s, reg)

  Retrieve value of signal 's' into 'reg'.

- #define **_setPreInit**
- #define **setPre**
- #define **freezePre**
- #define **freezePreClear**
- #define PRESENTPRE(s)

  Test for presence of signal in previous tick.

- #define PRESENTPREELSE(s, label)

  Test for presence of signal in previous tick.

- #define VALPRE(s)

  Retrieve previous value of signal 's'.

- #define VALPREREG(s, reg)

  Retrieve previous value of signal 's' into 'reg'.

- #define GOTO(label)

  Just a goto that also gets counted as instruction.

- #define ISAT(id, statelabel)

  Test whether an Exit Action has to be performed (predicate).

- #define ISATELSE(id, statelabel, label)

  Test whether an Exit Action has to be performed (control flow operation).

- #define CALL(label)

  Call a function at 'label'.

- #define RET

  Return from a function call.

- #define ISATCALL(id, l_state, l_call)

  Conditionally call a function.

- #define instrCntIncr

  Instruction counting/Tracing.

- #define **instrCntIncrc**
- #define **instrCntDecr**
- #define trace0(f) printf(f);

  If tracing is turned on, print trace string.

- #define **trace1**(f, a) printf(f, a);
- #define **trace2**(f, a, b) printf(f, a, b);
- #define **trace3**(f, a, b, c) printf(f, a, b, c);
- #define **trace4**(f, a, b, c, d) printf(f, a, b, c, d);
- #define **trace5**(f, a, b, c, d, e) printf(f, a, b, c, d, e);
- #define **trace6**(f, a, b, c, d, e, g) printf(f, a, b, c, d, e, g);
- #define **trace7**(f, a, b, c, d, e, g, h) printf(f, a, b, c, d, e, g, h);
- #define **trace8**(f, a, b, c, d, e, g, h, i) printf(f, a, b, c, d, e, g, h, i);
- #define **trace0c**(f) printf(f),
- #define **trace1c**(f, a) printf(f, a),
- #define **trace2c**(f, a, b) printf(f, a, b),
- #define **trace3c**(f, a, b, c) printf(f, a, b, c),
- #define **elsetrace** else {
- #define **elsetraceend** }
- #define traceThread(s)

  Count instruction (optionally), print trace string prefix (optionally).

- #define **traceThreadc**(s)
- #define trace0t(s, f) traceThread(s) trace0(f)

  Print trace prefix + suffix.

- #define **trace1t**(s, f, a) traceThread(s) trace1(f, a)
- #define **trace2t**(s, f, a, b) traceThread(s) trace2(f, a, b)
- #define **trace3t**(s, f, a, b, c) traceThread(s) trace3(f, a, b, c)
- #define **trace4t**(s, f, a, b, c, d) traceThread(s) trace4(f, a, b, c, d)
- #define **trace5t**(s, f, a, b, c, d, e) traceThread(s) trace5(f, a, b, c, d, e)
- #define **trace6t**(s, f, a, b, c, d, e, f1) traceThread(s) trace6(f, a, b, c, d, e, f1)
- #define **trace7t**(s, f, a, b, c, d, e, f1, g) traceThread(s) trace7(f, a, b, c, d, e, f1, g)
- #define **trace8t**(s, f, a, b, c, d, e, f1, g, h) traceThread(s) trace7(f, a, b, c, d, e, f1, g, h)
- #define **trace0tc**(s, f) traceThreadc(s) trace0c(f)
- #define **trace1tc**(s, f, a) traceThreadc(s) trace1c(f, a)
- #define **trace2tc**(s, f, a, b) traceThreadc(s) trace2c(f, a, b)
- #define **trace3tc**(s, f, a, b, c) traceThreadc(s) trace3c(f, a, b, c)
- #define **_SC_ERROR_NONE** 0
- #define **_SC_ERROR_NORESET** 1
- #define **_SC_ERROR_PRIORITY** 2
- #define **_SC_ERROR_CAUSALITY** 3
- #define _SC_ERROR0(code, f) fprintf(stderr, f); exit(code);

  Exit on errors.

- #define **_SC_ERROR1**(code, f, a) fprintf(stderr, f, a); exit(code);
- #define **_SC_ERROR2**(code, f, a, b) fprintf(stderr, f, a, b); exit(code);
- #define **_SC_ERROR3**(code, f, a, b, c) fprintf(stderr, f, a, b, c); exit(code);

Typedefs

- typedef int labeltype
  switch/case
- typedef unsigned int bitvector
  32 bits on IA32
- typedef bitvector signalvector
  32 signals on IA32
- typedef int threadtype
  Thread id/priority.
- typedef bitvector threadvector
  32 threads on IA32

Functions

- void getInputs ()
  Initialize signals to inputs for one tick.
- int checkOutputs (signalvector *tickOutputs)
  Set reference outputs and check valued signals, if there are any.
- void printVal (int id)
  Print value of a signal, if it has one.
- int tick ()
  Compute one tick.
- void selectCid ()
  Functions defined in sc.c.

Variables

- signalvector signals
  Bit mask for signals.
- threadvector enabled
  Bit mask for enabled threads.
- threadvector active
  Bit mask for active threads.
- int runCnt
  Counts program runs.

- int tickCnt

  Counts program ticks.

- int tickInstrCnt

  Instructions in one tick.

- int _notInitial

  Flag that indicates not-initial tick.

- threadtype _cid

  Id of current thread.

- labeltype _pc [_idMax]

  Pseudo program counters.

- threadvector _descs [_idMax]

  Descendants of thread.

- threadtype _parent [_idMax]

  Parent of thread.

- labeltype _returnAddress

  For function calls (eg Exit Actions).

- signalvector _presence_tested

  Signals that have been checked for presence in current tick.

- char * statePrev [_idMax]

  State where thread resumed previous tick.

- char * state [_idMax]

  State where thread resumed current tick.

- int runMax

  # of runs to execute

- int tickMax

  # of ticks to execute

- const char * s2signame [ ]

  Names of signals.

2.2.1 Detailed Description

Definition of SyncChart C macros.

See README.txt for general information. See LICENSE.txt for licensing information. For further information, see http://www.informatik.uni-kiel.de/rtsys/sc/ .

Author:

Reinhard v. Hanxleden, rvh@informatik.uni-kiel.de

2.2.2 Define Documentation

2.2.2.1 #define _BEGIN_SWITCH

Value:

```
while (1) {                                     \
                            _L_SWITCH: switch (_state) {                \
                             case _L_INIT:
```

2.2.2.2 #define _checkEMIT(s)

Value:

```
if (_presence_tested & u2b(s)) {                                        \
  _SC_ERROR2(_SC_ERROR_CAUSALITY,                                       \
             "SC ERROR (Causality): Signal %s/%d emitted after test for presence!\n",\
             s2signame[s], s)                                           \
  }
```

2.2.2.3 #define _CONCAT_helper(a, b) a ## b

Construct a label, of the form "_L'line in source file'".

Origindally contributed by Nicolas Berthier (nicolas.berthier@imag.fr)

To avoid label clashes, one must

- not generate multiple labels at the same line (this could be, eg, "PAUSE; PAUSE" in one line)
- not include another files within a function (this appears unlikely anyway)

Note that goto labels have function scope, hence it is ok to have identical labels in different files, as long as they belong to different functions.

Note also that the ## preprocessing macro, which concatenates strings, prevents macro expansion of it arguments. Thus the construction using _CONCAT and _CONCAT_helper.

2.2.2.4 #define _declState static labeltype _state;

Dispatcher.

When using switch-case logic, embed the tick function into a switch statement, in turn embedded in an infinite while loop.

2.2.2.5 #define _SC_ERROR0(code, f) fprintf(stderr, f); exit(code);

Exit on errors.

code: error code f: format string for error message a, b, ...: arguments for f

2.2.2.6 #define _SC_ERROR_DETECT_NORESET

Value:

```
if (_notInitial != 12345678) {                                  \
  _SC_ERROR0(_SC_ERROR_NORESET,                                 \
 "SC ERROR (Missing Reset): RESET() must be called before initial tick!\n");\
  }
```

Start a tick (an instant). 'p' denotes the main thread.

IF this is the initial tick ('_notInitial' is not set), THEN initialize things and continue with following instruction, ELSE call dispatcher to resume where we left off.

This also initializes the _TickEnd thread. Note that _parent[_TickEnd] is undefined. Note also that _descs[_TickEnd] does not matter, as that thread should never perform an ABORT (TRANS) or JOIN.

The flag _notInitial is set to some exotic value ("12345678") to catch (most) cases where an automaton reset has been forgotten.

2.2.2.7 #define _SC_ERROR_DETECT_PRIO(p)

Value:

```
if (isEnabled(p)) {                                             \
    _SC_ERROR1(_SC_ERROR_PRIORITY,                              \
               "SC ERROR (Priority Uniqueness): Priority %d already in use!\n", \
               p)                                               \
  }
```

Check whether PRIO tries to acquire priority that is already in use.

2.2.2.8 #define ABORT

Value:

```
do {                                                    \
    ABORT_;                                             \
    trace2t("ABORT:", "disables 0%o, enabled = 0%o\n",  \
            _descs[_cid], enabled)                      \
  } while (0)
```

Abort (terminate) descendant threads.

2.2.2.9 #define ABORT_

Value:

```
disableSet(_descs[_cid]);                               \
    deactivateSet(_descs[_cid])
```

Helper function (if/else-unsafe).

2.2.2.10 #define AWAIT(s)

Value:

```
do {                                                    \
    trace0t("AWAIT:", "initial pause\n")                \
    goto __LABELL__;                                    \
    _case __LABEL__:    _checkPRESENT(s)                \
    if (!(signals & u2b(s))) {                          \
      trace2t("AWAIT:", "determines %s/%d absent, waits\n",  \
            s2signame[s], s)                            \
        __LABELL__: PAUSEG_(__LABEL__);                 \
    }                                                   \
    trace2t("AWAIT:", "determines %s/%d present, proceeds\n", s2signame[s], s) \
  } while (0)
```

Await (non-immediately) signal 's'.

Pause; Then, IF 's' is present, THEN proceed to next instruction, ELSE pause.

Shorthand for 'PAUSE; AWAITI(s)', or, alternatively: 'elselabel: PAUSE; PRESENT(s, elselabel)'.

2.2.2.11 #define AWAITI(s)

Value:

```
do {                                                    \
    _case __LABEL__:    _checkPRESENT(s)                \
    if (!(signals & u2b(s))) {                          \
      trace2t("AWAITI:", "determines %s/%d absent, waits\n",  \
            s2signame[s], s)                            \
      PAUSEG_(__LABEL__);                               \
    }                                                   \
    trace2t("AWAITI:", "determines %s/%d present, proceeds\n", s2signame[s], s) \
  } while (0)
```

Await (immediately) signal 's'.

IF 's' is present, THEN proceed to next instruction, ELSE pause.

Shorthand for 'GOTO(label); elselabel: PAUSE; label: PRESENT(s, elselabel)'.

2.2.2.12 #define CALL(label)

Value:

```
do {                                            \
    trace1t("CALL:", "calls %s\n", #label)      \
      _returnAddress = _ref(__LABEL__);         \
    _goto(label);                               \
  _case __LABEL__: (void) 0;                        \
  } while (0)
```

Call a function at 'label'.

Use this if an Exit Action _must_ be performed.

2.2.2.13 #define clearPC(id)

Value:

```
statePrev[id] = "_L_INIT";                                          \
  state[id] = "_L_INIT"
```

2.2.2.14 #define dispatch_ goto _L_DISPATCH

If _SC_INLINE_DISPATCH is defined, call dispatcher at each operator that needs it. Otherwise, create shared code block for TERM/PAUSE/dispatch.

This can be included by TICKEND

2.2.2.15 #define EMIT(s)

Value:

```
do {                                                        \
    trace2t("EMIT:", "emits %s/%d\n", s2signame[s], s)              \
    _checkEMIT(s)                                                   \
    signals |= u2b(s);                                              \
  } while (0)
```

Emission of a pure signal 's'.

2.2.2.16 #define EMITINT(s, val)

Value:

```
do {                                            \
    valSigInt[s] = val;                                             \
    trace3t("EMITINT:", "emits %s/%d, value %d\n",                  \
            s2signame[s], s, val)                                   \
    _checkEMIT(s)                                                   \
    signals |= u2b(s);                                              \
  } while (0)
```

Emission of a valued signal 's', type integer.

2.2.2.17 #define EMITINTADD(s, val)

Value:

```
do {                                    \
    valSigInt[s] += val;                                            \
    trace4t("EMITINTADD:", "emits %s/%d, value %d, result %d\n",                \
            s2signame[s], s, val, valSigInt[s])                 \
    _checkEMIT(s)                                                   \
    signals |= u2b(s);                                              \
  } while (0)
```

Emission of a valued signal 's', type integer, combined with + .

2.2.2.18 #define EMITINTMAX(s, val)

Value:

```
do {                                    \
    valSigInt[s] = MAX(val, valSigInt[s]);                                          \
    trace4t("EMITINTMAX:", "emits %s/%d, value %d, result %d\n",                \
            s2signame[s], s, val, valSigInt[s])                 \
    _checkEMIT(s)                                                       \
    signals |= u2b(s);                                                  \
  } while (0)
```

Emission of a valued signal 's', type integer, combined with the maximum function .

2.2.2.19 #define EMITINTMIN(s, val)

Value:

```
do {                                    \
    valSigInt[s] = MIN(val, valSigInt[s]);                                          \
    trace4t("EMITINTMIN:", "emits %s/%d, value %d, result %d\n",                \
            s2signame[s], s, val, valSigInt[s])                 \
    _checkEMIT(s)                                                       \
    signals |= u2b(s);                                                  \
  } while (0)
```

Emission of a valued signal 's', type integer, combined with the minimum function .

2.2.2.20 #define EMITINTMUL(s, val)

Value:

```
do {                                    \
    valSigInt[s] *= val;                                                \
    trace4t("EMITINTMUL:", "emits %s/%d, value %d, result %d\n",                \
            s2signame[s], s, val, valSigInt[s])                 \
    _checkEMIT(s)                                                       \
    signals |= u2b(s);                                                  \
  } while (0)
```

Emission of a valued signal 's', type integer, combined with $*$.

2.2.2.21 #define enable(id)

Value:

```
enabled |= u2b(id);                             \
  active  |= u2b(id)
```

Thread enabling/disabling.

2.2.2.22 #define enableInit(id)

Value:

```
enabled = u2b(id);                              \
  active  = enabled
```

2.2.2.23 #define FORK(label, p)

Value:

```
do {                                                \
    FORK_(label, p);                                                \
    trace3t("FORK:", "forks %d/%s, active = 0%o\n", p, #label, active)  \
  } while (0)
```

Spawn a thread at 'label', with priority 'p'.

2.2.2.24 #define FORK_(label, p)

Value:

```
initPC(p, label);                                               \
  _parent[p] = _cid;                                            \
  _forkdescs |= u2b(p);                                         \
  enable(p)
```

Helper function (if/else-unsafe).

2.2.2.25 #define FORKE(label)

Value:

```
do {                                            \
  trace1t("FORKE:", "continues at %s\n", #label)                \
    FORKE_(label);                                              \
  } while (0)
```

Denote parent thread, starting at 'label'.

Must also calculate descendants. Descendants are used

- to check for termination (with JOIN)
- to be disabled upon abortion (with TRANS)

2.2.2.26 #define FORKE_(label)

Value:

```
setPC(_cid, label);                                             \
  _descs[_cid] = _forkdescs;                                    \
  _forkdescs = 0;                                               \
  _pid = _cid;                                                  \
  while ((_ppid = _parent[_pid]) != _TickEnd)   {               \
    _descs[_ppid] |= _descs[_pid];                              \
    _pid = _ppid;                                               \
  }                                                             \
  dispatch_
```

Helper function (if/else-unsafe).

2.2.2.27 #define GOTO(label)

Value:

```
do {                                    \
    trace1t("GOTO:", "transfer to %s\n", #label)                \
    instrCntIncr                                                \
    _goto(label);                                               \
  } while (0)
```

Just a goto that also gets counted as instruction.

2.2.2.28 #define HALT

Value:

```
do {                                                    \
    _case __LABEL__: trace1t("HALT:", "pauses, active = 0%o\n", active) \
    PAUSEG_(__LABEL__);                                 \
  } while (0)
```

Shorthand for 'label: PAUSE; GOTO(label)'.

2.2.2.29 #define initPC(p, label)

Value:

```
_pc[p] = _ref(label);                                           \
  statePrev[p] = "_L_INIT";                                     \
  state[p] = #label
```

2.2.2.30 #define instrCntIncr

Instruction counting/Tracing.

Increment/decrement SC instruction counter.

Decrement is needed in some places to avoid duplicate counting.

2.2.2.31 #define ISAT(id, statelabel)

Value:

```
(trace2tc("ISAT:", "%s at %s\n",                                \
            (isEnabled(id) && (_pc[id] == _ref(statelabel))) ? "_is_" : "is _not_", #statelabel) \
   (isEnabled(id) && (_pc[id] == _ref(statelabel))))
```

Test whether an Exit Action has to be performed (predicate).

IF thread 'id' is active and at state 'statelabel', THEN return 1, ELSE return 0.

2.2.2.32 #define ISATCALL(id, l_state, l_call)

Value:

```
do {                              \
    if (isEnabled(id) && (_pc[id] == _ref(l_state))) {                  \
      trace1t("ISATCALL:", "calls %s\n", #l_call)                       \
      _returnAddress = _ref(__LABEL__);                                 \
      _goto(l_call);                                                    \
    }                                                                   \
    trace1t("ISATCALL:", "does _not_ call %s\n", #l_call)               \
    _case __LABEL__: (void) 0;                                          \
  } while (0)
```

Conditionally call a function.

IF thread 'id' is active and at state 'statelabel', THEN call function at 'label'; Use this if an Exit Action _may_ have to be performed Shorthand for 'ISATELSE(id, l_state, l); CALL(l_call); l:'

2.2.2.33 #define ISATELSE(id, statelabel, label)

Value:

```
{                                 \
  if (isEnabled(id) && (_pc[id] == _ref(statelabel))) {                 \
    trace1t("ISATELSE:", "_is_ at %s\n", #statelabel)                   \
  } else {                                                              \
    trace2t("ISATELSE:", "is _not_ at %s, transfer to %s\n",            \
            #statelabel, #label)                                        \
    _goto(label);                                                       \
  }}
```

Test whether an Exit Action has to be performed (control flow operation).

IF thread 'id' is active and at state 'statelabel', THEN proceed to next instruction, ELSE jump to 'label'. Shorthand for 'if (!ISAT(id, statelabel)) goto label'.

2.2.2.34 #define JOIN

Value:

```
do {                                                    \
    _case __LABEL__:                                                    \
    trace0t("JOIN:", isEnabledAnyOf(_descs[_cid]) ? "waits\n" : "joins\n") \
    if (isEnabledAnyOf(_descs[_cid])) {                                 \
      PAUSEG_(__LABEL__);                                               \
    }                                                                   \
  } while (0)
```

Shorthand for 'elselabel: JOINELSE(elselabel)'. Join completed child threads.

IF all descendants have terminated, THEN proceed, ELSE pause, resume at JOIN.

2.2.2.35 #define JOINELSE(elselabel)

Value:

```
do {                                    \
  JOINELSEG_(__LABEL__, elselabel);                                     \
_case __LABEL__: (void) 0;                                              \
  } while (0)
```

Join completed child threads.

IF all descendants have terminated, THEN proceed after JOINELSE, ELSE pause, resume at 'elselabel'.

Semantically, this is the primitive Join-operator, from which the others can be derived; hence JOINELSE appears first, and the other operators are considered shorthands.

In terms of implementation, the operators are built from JOINELSEG, which semantically corresponds to JOINELSE + GOTO.

2.2.2.36 #define JOINELSEG(thenlabel, elselabel)

Value:

```
do {                            \
    JOINELSEG_(thenlabel, elselabel);                                   \
  } while (0)
```

Shorthand for 'JOINELSE(elselabel); GOTO(thenlabel)'.

IF all descendants have terminated, THEN jump to 'thenlabel', ELSE pause, resume at 'elselabel'

2.2.2.37 #define JOINELSEG_(thenlabel, elselabel)

Value:

```
if (isEnabledNoneOf(_descs[_cid])) {                                    \
    trace1t("JOINELSEG:", "joins, transfers to %s\n", #thenlabel)       \
    _goto(thenlabel);                                                   \
  }                                                                     \
  trace1t("JOINELSEG:", "does not join, pauses at %s\n", #elselabel)    \
  instrCntDecr                                                          \
  PAUSEG_(elselabel)
```

Helper function (if/else-unsafe).

2.2.2.38 #define JPPAUSE(p, elselabel)

Value:

```
do {                                    \
    trace2t("JPPAUSE:", "%s, prio = %d\n",                              \
            isEnabledNoneOf(_descs[_cid]) ? "joins" : "does not join", p) \
    JPPAUSEG_(p, __LABEL__, elselabel);                                 \
    _case __LABEL__: (void) 0;                                          \
  } while (0)
```

Shorthand for 'JOINELSE(label); GOTO thenlabel; label: PRIO(p); PAUSE; GOTO(elselabel); thenlabel:'.

IF all descendants have terminated, THEN proceed, ELSE set priority to 'p', pause, and continue at 'elselabel'.

This shorthand avoids the context switch immediately before the PAUSE.

2.2.2.39 #define JPPAUSEG(p, thenlabel, elselabel)

Value:

```
do {                              \
    trace2t("JPPAUSEG:", "%s, prio = %d\n",                              \
            isEnabledNoneOf(_descs[_cid]) ? "joins" : "does not join", p) \
    JPPAUSEG_(p, thenlabel, elselabel);                                  \
  } while (0)
```

Efficient shorthand for 'JOINELSE(label); GOTO thenlabel; label: PRIO(p); PAUSE; GOTO(elselabel)'.

IF all descendants have terminated, THEN jump to 'thenlabel', ELSE set priority, pause, and continue at 'elselabel'.

This shorthand avoids the context switch immediately before the PAUSE.

2.2.2.40 #define JPPAUSEG_(p, thenlabel, elselabel)

Value:

```
if (isEnabledNoneOf(_descs[_cid])) {                                     \
    _goto(thenlabel); }                                                  \
  instrCntDecr                                                           \
  PPAUSEG_(p, elselabel)
```

Helper function (if/else-unsafe).

2.2.2.41 #define mergedDispatch

Value:

```
_L_TERM:   disable(_cid);                                                \
_L_PAUSEG:   deactivate(_cid);                                           \
_L_DISPATCH: dispatch();
```

2.2.2.42 #define PAUSE

Value:

```
do {                                                                     \
    trace1t("PAUSE:", "pauses, active = 0%o\n", active)                  \
    PAUSEG_(__LABEL__); _case __LABEL__: (void) 0;                       \
  } while (0)
```

Pause a thread, resume at subsequent statement.

Semantically, this is the primitive operator. In terms of implementation, it is built with PAUSEG.

2.2.2.43 #define PAUSEG(label)

Value:

```
do {                                                \
    trace1t("PAUSEG:", "pauses, active = 0%o\n", active)                \
    PAUSEG_(label);                                                     \
  } while (0)
```

Shorthand for 'PAUSE; GOTO(label)'.

Pause a thread, resume at 'label'.

2.2.2.44 #define PAUSEG_(label)

Value:

```
setPC(_cid, label);                                                     \
    goto _L_PAUSEG
```

Helper function (if/else-unsafe).

2.2.2.45 #define PPAUSE(p)

Value:

```
do {                                                    \
    trace1t("PPAUSE:", "sets prio to %d, pauses\n", p)                  \
    PPAUSEG_(__LABEL__);                                                \
    _case __LABEL__: (void) 0;                                          \
  } while (0)
```

Efficient shorthand for 'PRIO(p); PAUSE'.

Set a priority, then pause (this sets "prionext").

This shorthand avoids the context switch immediately before the PAUSE.

2.2.2.46 #define PPAUSEG(p, label)

Value:

```
do {                                                \
    trace2t("PPAUSEG:", "sets prio to %d, pauses, resumes at %s\n", p, #label) \
    PPAUSEG_(p, label);                                                 \
  } while (0)
```

Efficient shorthand for 'PRIO(p); PAUSE; GOTO(label)'.

Set a priority, then pause (this sets "prionext"), resume at 'label'.

This shorthand avoids the context switch immediately before the PAUSE.

2.2.2.47 #define PPAUSEG_(p, label)

Value:

```
PRIO_(p);                                               \
  instrCntDecr                                          \
  PAUSEG_(label)
```

Helper function (if/else-unsafe).

2.2.2.48 #define PRESENT(s)

Value:

```
(trace3tc("PRESENT:", "determines %s/%d %s\n",                  \
          s2signame[s], s, (signals & u2b(s)) ? "present" : "absent")  \
   (_checkPRESENTc(s) signals & u2b(s)))
```

Test for presence of signal 's' (predicate).

IF 's' is present, THEN return 1, ELSE return 0.

2.2.2.49 #define PRESENTELSE(s, label)

Value:

```
do {                                    \
    _checkPRESENT(s)                                                \
    if (!(signals & u2b(s))) {                                      \
      trace3t("PRESENTELSE:", "determines %s/%d absent, transfers to %s\n", \
              s2signame[s], s, #label)                              \
      _goto(label);                                                 \
    }                                                               \
    trace2t("PRESENTELSE:", "determines %s/%d present\n", s2signame[s], s) \
  } while (0)
```

Test for presence of signal 's' (control flow op).

IF 's' is present, THEN proceed to next instruction, ELSE jump to 'label'

2.2.2.50 #define PRESENTEMIT(s, t)

Value:

```
do {                                            \
    _checkPRESENT(s)                                                \
    if (signals & u2b(s)) {                                         \
      trace4t("PRESENTEMIT:", "determines %s/%d present, emits %s/%d\n", \
              s2signame[s], s, s2signame[t], t)                     \
      _checkEMIT(t)                                         \
      signals |= u2b(t);                                            \
    }                                                               \
    elsetrace                                                       \
      trace2t("PRESENTEMIT:", "determines %s/%d absent\n", s2signame[s], s) \
    elsetraceend                                                    \
  } while (0)
```

If signal 's' is present, emit 't'.

Shorthand for 'PRESENTELSE(s, label); EMIT(t); label:'

2.2.2.51 #define PRESENTPRE(s)

Value:

```
(trace3tc("PRESENTPRE:", "determines %s/%d %s\n",                       \
          s2signame[s], s, (sigsPre & u2b(s)) ? "present" : "absent")  \
   (sigsPre & u2b(s)))
```

Test for presence of signal in previous tick.

IF 's' was present in previous tick, THEN return 1, ELSE return 0.

2.2.2.52 #define PRESENTPREELSE(s, label)

Value:

```
do {                                    \
    if (!(sigsPre & u2b(s))) {                                          \
      trace3t("PRESENTPRE:", "determines previous %s/%d absent, transfers to %s\n", \
              s2signame[s], s, #label)                                  \
      _goto(label);                                                     \
    }                                                                   \
    trace2t("PRESENTPRE:", "determines previous %s/%d present\n",       \
            s2signame[s], s)                                            \
  } while (0)
```

Test for presence of signal in previous tick.

IF 's' was present in previous tick, THEN proceed to next instruction, ELSE jump to 'label'

2.2.2.53 #define PRIO(p)

Value:

```
do {                                                    \
    trace1t("PRIO:", "set to priority %d\n", p)                         \
    PRIOG_(p, __LABEL__);                                               \
    _case __LABEL__: (void) 0;                                          \
  } while (0)
```

Change priority of a thread, from '_cid' to 'p'.

Semantically, this is the primitive operator. In terms of implementation, PRIOG is the basic operator.

2.2.2.54 #define PRIO_(p)

Value:

```
if (p != _cid) {                                                        \
    _SC_ERROR_DETECT_PRIO(p);                                           \
    _parent[p] = _parent[_cid];                                         \
    _pid = 0;                                                           \
    for (_ppid = _descs[_cid]; _ppid > 0; _ppid >>= 1) {                \
      if (_parent[_pid] == _cid)                                        \
        _parent[_pid] = p;                                              \
      _pid++;                                                           \
    }                                                                   \
    _descs[p] = _descs[_cid];                                           \
    _pid = _cid;                                                        \
    while ((_ppid = _parent[_pid]) != _TickEnd) {                       \
      _descs[_ppid] &= ~u2b(_cid);                                      \
      _descs[_ppid] |= u2b(p);                                          \
      _pid = _ppid;                                                     \
    }                                                                   \
    deactivate(_cid);                                                   \
    disable(_cid);                                                      \
    _cid = p;                                                           \
```

```
    enable(_cid);                                                   \
  }
```

Helper function (if/else-unsafe) to change priority of a thread, from '_cid' to 'p'.

IF p == _cid, THEN we do not have to do anything. ELSE:

IF p is already in use by another thread, THEN report an error. ELSE:

// Update parent pointers FOR ALL descendants _pid of current thread with _parent[_pid] == _cid: Change _parent[_pid] to p

// Update descendant lists FOR parent _ppid of current thread: Delete _cid from _descs[_ppid] Add p to _descs[_ppid] REPEAT recursively for parent of _ppid, until _TickEnd (root parent) is reached

Deactivate and disable _cid Set _cid = p Enable (and hence implicitly activate) _cid

2.2.2.55 #define PRIOG(p, label)

Value:

```
do {                                                \
    trace2t("PRIOG:", "set to priority %d, goto %s\n", p, #label)       \
    PRIOG_(p, label);                                                   \
  } while (0)
```

Shorthand for 'PRIO(p); GOTO(label)'.

2.2.2.56 #define PRIOG_(p, label)

Value:

```
PRIO_(p);                                                               \
  setPC(_cid, label);                                                   \
  dispatch_
```

Helper function (if/else-unsafe).

2.2.2.57 #define RESET()

Value:

```
do {                                    \
    trace0t("RESET:", "reset automaton\n")              \
    _notInitial = 0;                                    \
    tickCnt = 0;                                        \
  } while (0)
```

Reset automaton. Should be called before first call of tick function.

2.2.2.58 #define RET

Value:

```
do {                                                    \
    trace0t("RET:", "returns\n")                        \
    _goto(_deref(_returnAddress));                      \
  } while (0)
```

Return from a function call.

2.2.2.59 #define setPC(id, label)

Value:

```
_pc[id] = _ref(label);                                  \
  statePrev[id] = state[id];                            \
  state[id] = #label
```

2.2.2.60 #define SIGNAL(s)

Value:

```
do {                                            \
    trace2t("SIGNAL:", "initializes %s/%d\n", s2signame[s], s)  \
    signals &= ~u2b(s);                                 \
  } while (0)
```

Initialize a local signal (handles reincarnation).

2.2.2.61 #define SUSPEND(cond)

Value:

```
do {                                            \
    _case __LABEL__: if (cond) {                        \
    trace1t("SUSPEND:", "suspends itself and descendants 0%o\n", _descs[_cid]) \
    active &= ~_descs[_cid];                            \
    freezePre                                           \
    instrCntDecr                                        \
    PAUSEG_(__LABEL__);                                 \
  }                                                     \
 } while (0)
```

Suspend current thread if 'cond' is true.

Note: suspension is implemented by deactivating the current thread as well as its descendants. This exploits that the PCs of the descendants must reside at tick boundaries.

2.2.2.62 #define SUSPENDG(label)

Value:

```
do {                                            \
    trace1t("SUSPENDGOT:", "suspends itself and descendants 0%o\n", _descs[_cid]) \
    active &= ~_descs[_cid];                            \
    freezePre                                           \
    instrCntDecr                                        \
    PAUSEG_(label);                                     \
  } while (0)
```

Suspend current thread, goto 'label'.

Note: suspension is implemented by deactivating the current thread as well as its descendants. This exploits that the PCs of the descendants must reside at tick boundaries.

2.2.2.63 #define SUSTAIN(s)

Value:

```
do {                                                    \
    _case __LABEL__: trace2t("SUSTAIN:", "emits %s/%d\n", s2signame[s], s) \
    EMIT(s); PAUSEG_(__LABEL__);                                          \
  } while (0)
```

Sustain a pure signal 's'.

2.2.2.64 #define TERM

Value:

```
do {                                                    \
    trace1t("TERM:", "terminates, enabled = 0%o\n", enabled & ~u2b(_cid))       \
    TERM_;                                                                \
  } while (0)
```

Terminate a thread.

2.2.2.65 #define TICKEND

Value:

```
TERM_;                                          \
  _case _L_TICKEND: setPre                              \
  return isEnabledNotOnly(_TickEnd);                    \
  _END_SWITCH1                                          \
  mergedDispatch                                        \
  _END_SWITCH2
```

Complete a tick.

Return 0 iff computation has terminated. This starts with TERM_, to properly terminate a thread that runs into TICKEND.

2.2.2.66 #define TICKSTART(p)

Value:

```
static threadtype _pid, _ppid;                                  \
  static threadvector _forkdescs = 0;                                   \
  _declindex                                                            \
  _declState                                                            \
  freezePreClear                                                        \
  _checkTickInit                                                        \
  if (_notInitial) {                                                    \
    _SC_ERROR_DETECT_NORESET                                            \
```

```
    active = enabled;                                                   \
    dispatch_;                                                          \
  } else {                                                              \
    initPC(_TickEnd, _L_TICKEND);                                       \
    enableInit(_TickEnd);                                               \
    _cid = p;                                                           \
    _parent[_cid] = _TickEnd;                                           \
    clearPC(_cid);                                                      \
    enable(_cid);                                                       \
    _setStateInit                                                       \
    _setPreInit                                                         \
    _setValInit                                                         \
    _notInitial = 12345678;                                             \
  }                                                                     \
  _BEGIN_SWITCH
```

2.2.2.67 #define trace0t(s, f) traceThread(s) trace0(f)

Print trace prefix + suffix.

s is string denoting instruction (eg, "PAUSE:") f is format string for trace suffix a, b, ... are arguments for format string

2.2.2.68 #define traceThread(s)

Value:

```
instrCntIncr                                                            \
  trace3("%-9s %d/%s ", s, _cid, state[_cid])
```

Count instruction (optionally), print trace string prefix (optionally).

Trace string prefix takes a string s (typically denoting the instruction) and identifies the executing thread, both by name and thread id.

2.2.2.69 #define traceThreadc(s)

Value:

```
instrCntIncrc                                                           \
  trace3c("%-9s %d/%s ", s, _cid, state[_cid])
```

2.2.2.70 #define TRANS(label)

Value:

```
do {                                            \
    ABORT_;                                                             \
    trace3t("TRANS:", "disables 0%o, transfers to %s, enabled = 0%o\n", \
            _descs[_cid], #label, enabled)                              \
    _goto(label);                                                       \
  } while (0)
```

Transition to 'label', kill descendant threads (implements abortion).

Shorthand for 'ABORT; GOTO(label)'.

2.2.2.71 #define u2b(u) (1 << u)

Encoding of signal/thread 'u' (some non-negative int) in bitvector.

This implementation is fast and simple, BUT limits the max thread ID and max signal ID to the word width of the machine (eg 32).

2.2.2.72 #define VAL(s)

Value:

```
(                                                       \
    trace3tc("VAL:", "determines value of %s/%d as %d\n",              \
            s2signame[s], s, valSigInt[s])                              \
    valSigInt[s])
```

Retrieve value of signal 's'.

2.2.2.73 #define VALPRE(s)

Value:

```
(                                                \
    trace3tc("VALPRE:", "determines value of %s/%d as %d\n",    \
            s2signame[s], s, valSigIntPre[s])                   \
    valSigIntPre[s])
```

Retrieve previous value of signal 's'.

2.2.2.74 #define VALPREREG(s, reg)

Value:

```
do {                            \
    trace3t("VALPREREG:", "determines value of %s/%d as %d\n",  \
            s2signame[s], s, valSigIntPre[s])                   \
    reg = valSigIntPre[s];                                      \
  } while (0)
```

Retrieve previous value of signal 's' into 'reg'.

2.2.2.75 #define VALREG(s, reg)

Value:

```
do {                                            \
    trace3t("VALREG:", "determines value of %s/%d as %d\n",             \
            s2signame[s], s, valSigInt[s])                              \
    reg = valSigInt[s];                                                 \
  } while (0)
```

Retrieve value of signal 's' into 'reg'.

2.2.3 Function Documentation

2.2.3.1 void selectCid ()

Functions defined in sc.c.

Functions defined in sc.c.

Uses obvious algorithm, run time linear in position of highest bit. Note that there are also alternatives that run logarithmic to bit vector size. See eg http://graphics.stanford.edu/~seander/bithacks.html#IntegerLog . Which is actually faster depends on application.

2.2.3.2 int tick ()

Compute one tick.

Returns 1 if some thread is still active in current tick.

2.2.4 Variable Documentation

2.2.4.1 char* statePrev[_idMax]

State where thread resumed previous tick.

Check whether _SC_NOTRACE has been defined (eg from gcc command line). If so, suppress tracing and instruction counting. This then results in compact macro-expanded source code and executable.

2.3 selectCid.c File Reference

```
#include "sc.h"
```

Functions

- void selectCid ()

  Computing the id of next thread to be dispatched.

2.3.1 Detailed Description

Auxiliary file for using SyncChart C macros. selectCid.c is a fragment of sc.c. It should be linked in by programs that do not need the main() function provided by sc.c, such as PCO.

See README.txt for general information. See LICENSE.txt for licensing information. For further information, see http://www.informatik.uni-kiel.de/rtsys/sc/ .

Author:

Reinhard v. Hanxleden, rvh@informatik.uni-kiel.de

2.3.2 Function Documentation

2.3.2.1 void selectCid ()

Computing the id of next thread to be dispatched.

Functions defined in sc.c.

Uses obvious algorithm, run time linear in position of highest bit. Note that there are also alternatives that run logarithmic to bit vector size. See eg http://graphics.stanford.edu/~seander/bithacks.html#IntegerLog . Which is actually faster depends on application.

Index